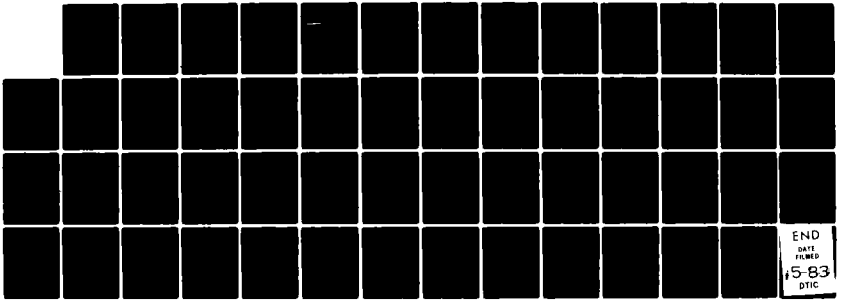
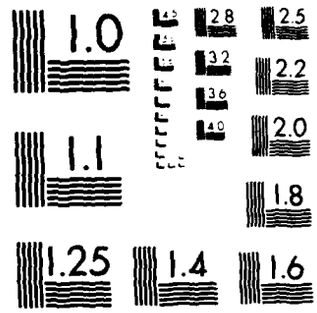
AD-A127 576 IMPLEMENTING SPECIFICATION FREEDOMS(U) UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFORMATION SCIENCES INST M FEATHER ET AL. APR 83 ISI/RR-83-100 1/1

UNCLASSIFIED MDA903-81-C-0335 F/G 9/2 NL



END
DATE FILMED
5-83
DTIC



(12)

ISI/RR-83-100
April 1983

University of Southern California

Martin Feather
Philip London

Implementing Specification Freedoms

83 05 03 018

INFORMATION SCIENCES INSTITUTE

213/822-1511
4676 Admiralty Way/Marina del Rey/California 90291-6695

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1 REPORT NUMBER ISI/RR-83-100	2 GOVT ACCESSION NO. AD-A127 576	3 RECIPIENT'S CATALOG NUMBER
4 TITLE (and Subtitle) Implementing Specification Freedoms		5 TYPE OF REPORT & PERIOD COVERED Research Report 6 PERFORMING ORG. REPORT NUMBER
7 AUTHOR(s) Martin Feather and Philip London		8 CONTRACT OR GRANT NUMBER(s) MDA903 81 C 0335
9 PERFORMING ORGANIZATION NAME AND ADDRESS USC/Information Sciences Institute 4676 Admiralty Way Marina del Rey, CA 90291		10 PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS .....
11 CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12 REPORT DATE April 1983 13 NUMBER OF PAGES 54
14 MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15 SECURITY CLASS. (of this report) Unclassified 15a DECLASSIFICATION DOWNGRADING SCHEDULE

16 DISTRIBUTION STATEMENT (of this Report)

This document is approved for public release and sale; distribution is unlimited.

17 DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

.....

18 SUPPLEMENTARY NOTES

Current address of Philip London: Teknowledge
525 University Avenue
Palo Alto, CA 94301

19 KEY WORDS (Continue on reverse side if necessary and identify by block number)

automated software development, formal specification, Gist, program modification, program transformation, very high-level languages

20 ABSTRACT (Continue on reverse side if necessary and identify by block number)

(OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (continued)

The process of converting formal specifications into valid implementations is central in the development of reliable software. As formal specification languages are enriched with constructs to enhance their expressive capabilities and as they increasingly afford specificational freedoms by requiring only a description of intended behavior rather than a prescription of particular algorithms, the gap between specification and implementation widens so that converting specifications into implementations becomes even more difficult. A major problem lies in the mapping of high-level specification constructs into an implementation that effects the desired behavior. In this report, we consider the issues involved in eliminating occurrences of high-level specification-oriented constructs during this process. We discuss mapping issues in the context of our development methodology, in which implementations are derived via the application of equivalence-preserving transformations to a specification language whose high-level expressive capabilities are modeled after natural language. After the general discussion, we demonstrate the techniques on a real system whose specification is written in this language.

ISI/RR-83-100

April 1983

University of Southern California

Martin Feather

Philip London

Implementing Specification Freedoms

INFORMATION SCIENCES INSTITUTE

213/822-1511

4676 Admiralty Way/Marina del Rey/California 90291-6695

This research is supported by the Defense Advanced Research Projects Agency under Contract No. MDA903 81 C 0335. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policy of DARPA, the U.S. Government, or any person or agency connected with them.

CONTENTS

ACKNOWLEDGMENTS

We wish to thank the other members of the ISI Transformational Implementation group: Bob Balzer, Wellington Chiu, Don Cohen, Lee Erman, Steve Fickas, Neil Goldman, Bill Swartout, and Dave Wile. Collectively they have defined the context within which this work lies, and individually they have improved this report through frequent discussions about the research and helpful comments on the drafts. Thanks also to Sheila Coyazo for editing the final draft. Portions of this report appeared in a paper of the same name published in *Science of Computer Programming* [21].

1. INTRODUCTION

As formal specification technology continues to develop, the constructs available in specification languages will differ increasingly from those available in the various implementation languages. A problem then arises in the mapping between these disparate language levels. Implementation languages simply do not possess the ability to directly express concepts found in specifications. This is as it should be, because the languages are designed for different purposes: implementation languages are for describing efficient algorithms, and specification languages are for describing behaviors.

One goal in the design of formal specification languages is to ease the task of writing specifications. One approach is to use a specification language that reduces the burden in two ways. First, by enhancing expressiveness, this type of language allows the specifier to state his desires more easily. Second, by requiring only a description of intended behavior rather than the detailed specification of a particular algorithm, this language affords a specifier the freedom to specify *what* is desired rather than *how* to achieve it.

This approach, however, aggravates the problem of producing a correct implementation from the design specification. In this paper, we investigate a solution to this problem by presenting a number of implementation options for each of several high-level specification constructs. We then demonstrate that the *mappings* of these high-level specification constructs into implementations are derivable by sequential application of relatively straightforward correctness-preserving[1] transformations. The collection of high-level specification mappings can be viewed as the major conceptual steps in a "transformational implementation."

The work presented here should be viewed as being but part of a larger effort (being conducted by the Transformational Implementation (TI) group at ISI) investigating reliable software development by considering methods for automating aspects of the software development process. The methodology we have adopted for developing reliable software comprises the following activities:

1. system specification in a formal language designed for specification;
2. elimination of high-level specification constructs by mechanical application of correctness-preserving transformations;
3. selection and development of algorithms and abstract data types to effect behavior described in the specification (also by mechanical application of correctness-preserving transformations); and
4. translation into the target implementation language.

The implementor maps the specification into an implementation through the selection and application of appropriate transformations (from a pre-existing catalog). The programmer in this scenario has control over the implementation process, making many of the same decisions he would ordinarily make. Our goal is to construct a system which will relieve some of the implementor's burden by performing the perfunctory tasks of bookkeeping and program source text maintenance. Specifically, the following portions of this software development system will be automated:

---

[1] Since in our view a specification denotes a set of behaviors, our notion of a "correct" transformation is one whose application results in a specification denoting a subset of those behaviors (a nonempty subset, provided that the original specification denoted a nonempty set of behaviors).

1. tools to assist the implementor in deciding on the appropriateness and applicability of a given transformation,
2. a mechanism for applying a chosen transformation to the developing program,
3. support for the development process (e.g., automatic production of documentation and "replay" facilities which allow a development to be repeated so as to reimplement a modified specification),
4. the catalog of correctness-preserving source-to-source transformations, which embodies the knowledge of alternative implementations of particular specificational constructs, and
5. a mechanism for translating a fully developed program into a target implementation language.

The specification language itself is critical within this software development framework. Two basic characteristics are required of the specification language. The first of these is that the language provide the flexibility and ease of expression required for describing the full range of acceptable behaviors of the system under design. See [2] for a description of the requirements for specification languages that will exhibit these characteristics. The second requirement of a specification language for our software development methodology is that it be wide-spectrum [7]. This means, in essence, that the same language can serve both as a specification language (for describing the full range of acceptable behaviors) and as an implementation language (for describing an efficient program whose behavior is true to the specification). In reality, the specification language need be wide-spectrum only up to a point; after selection of algorithms and abstract data types, the "implementation" can automatically be translated into a suitable implementation language.

Our group has developed such a language, called Gist [25], which permits expressibility by providing many of the constructs found in natural language specifications of processes. These expressive capabilities include *historical reference* (the ability to refer to past process states), *constraints* (restrictions on acceptable system behavior in the form of global declarations), a relational and associative data model which captures the logical structure without imposing an access regime, *inference* (which allows for global declarations describing relationships among data), and *demons* (asynchronous process responding to defined stimuli), among others. Thus, the effort in Gist has been to provide the specificational expressiveness of natural language while imposing formal syntax and semantics.[2]

This paper focuses on the transformations used to eliminate these high-level specification constructs. Such elimination is obviously necessary, because no target implementation language is expected to provide such facilities. More important, to the extent that the specification language is doing its job of describing intended behavior (*what*) without prescribing a particular algorithm (*how*), these constructs represent the freedoms offered by the specification language. How such freedoms are implemented determines in large part the efficiency of the resulting algorithm. Furthermore, as such constructs are just beginning to be incorporated into specification languages, consideration of alternative implementations of these constructs has received little attention. Absence of these constructs has forced systems analysts and designers to choose (normally unconsciously) one implementation as a precondition to expressing a specification.

---

[2]Note that Gist syntax is ALGOL-like and not English-like!

The development of these high-level transformations, which we call *mappings*, presents a rich set of issues dealing with the translation of specifications into implementations. As it is often difficult to separate the activities of mapping high-level constructs and selecting and developing algorithms and data types, the discussion of mapping transformations will also consider these issues.

2. PACKAGE ROUTER PROBLEM

2.1 OUTLINE OF PROBLEM

To illustrate our approach, we choose as an example a routing system for distributing packages into destination bins. This problem was constructed by representatives of the process control industry to be typical of their real-world applications. Hommel's study of various programming methodologies used this problem as the comparative example [26].

Figure 2-1 illustrates the routing network. At the top, a source station feeds packages one at a time into the network, which is a binary tree consisting of switches connected by pipes. The terminal nodes of the binary tree are the destination bins.

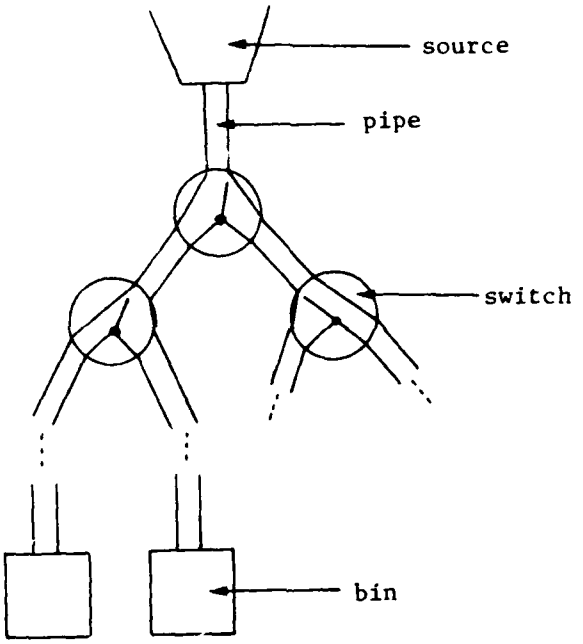


Figure 2-1: Package router

When a package arrives at the source station, its intended destination (one of the bins) is determined. The package is then released into the pipe leading from the source station. For a package to reach its designated destination bin, the switches in the network must be set to direct the package through the network and into the correct bin.

Packages move through the network by gravity (working against friction), and so steady movement of packages cannot be guaranteed; they may "bunch up" within the network and thus make it impossible to set a switch properly between the passage of two such bunched packages (a switch cannot be set when there is a package or packages in the switch for fear of damaging such packages). If a new package's destination differs from that of the immediately preceding package, its

release from the source station is delayed a (precalculated) fixed length of time (to reduce the chance of bunching). In spite of such precautions, packages may still bunch up and become misrouted, ending up in the wrong bin; the package router is to signal such an event.

Only a limited amount of information is available to the package router to effect its desired behavior. At the time of arrival at the source station, but not thereafter, the destination of a package may be determined. The only means of determining the locations of packages within the network is a group of sensors (placed on the entries and exits of switches and on the entries of bins); these sensors detect the passage of packages but are unable to determine their identity. (The sensors are able to recognize the passage of individual packages, regardless of bunching).

2.2 GIST USED TO SPECIFY PROBLEM

The specification task is to denote how the portion to be implemented (switches, source station) behaves. To accomplish this, the specification models not only this portion, but also the surrounding environment, to form a closed system. The environment is modeled in only as much detail as is necessary to express the properties needed by the system to be implemented (e.g., the rate of movement of packages through the router is unpredictable; however, packages never start moving backwards through the network, and packages never overtake one another). Within this closed system, the portions to be implemented are the controlling mechanism for the switches within the network and the source station releasing successive packages into the network.

In specifying the system, we aim to make a clear and correct statement of the behavior the switches and source station must exhibit in their interaction with each other and the environment. However, the specification strives to describe the behavior directly, without resorting to an algorithm that effects that behavior; deriving such an algorithm is rightly part of the separate task of implementing the specification. Therefore, the emphasis in specification is on *what* rather than *how*. This emphasis is important for making a clear distinction between specification and implementation; by describing *what*, we do not restrict the implementation freedoms available. Section 3 describes some of Gist's constructs that permit an easy statement of "what." The entire package router specification is presented in Appendix I.

In a further effort to simplify the specification of intended behavior, Gist specifications assume p[illegible]ct knowledge. That is, any information used to describe the behavior of a system is available to each component part, to describe its interactions with other parts. This assumption is often not satisfied in the actual system. In the package router example, the specification relies upon knowing the location and destination of each package. However, in the actual system the destination of a package is only accessible while it is at the source, and its location is only deducible from sensor data indicating the passage of packages through the network. In an implementation the unavailable information must be deduced from other, available information. This problem would complicate the description of the system behavior by substituting a *how* for a *what* description. For this reason, we have therefore separated these two issues. The specification is based on the perfect knowledge assumption and the actual implementation is described separately, as are all other implementation issues.

3. MAPPINGS

In this section we will consider in turn several high-level Gist constructs and show the following for each:

1. why it is a useful specification construct, illustrated by presenting a use of it in the package router problem;
2. what implementation alternatives exist, achieved by mapping the construct away, and what the implications are of those alternatives.

In Section 4 we will demonstrate how these mappings can be derived.

3.1 HISTORICAL REFERENCE

Historical reference refers to the ability to extract information from any preceding state in the computation history. The ability to do this frees the specification from determining in advance (and remembering) all current information that might be required at some time in the future. Required information is merely retrieved at the point of consumption without concern for when it is available.

3.1.1 Example From Package Router

The source station is to control the release of packages into the network as follows: If a newly arrived package's destination differs from that of the immediately preceding package, delay release of the new arrival, otherwise release it at once. The "historical reference" here is to the destination of the immediately preceding package. Using Gist, this portion of the specification is expressed as follows:

```
demon RELEASE_PACKAGE_INTO_NETWORK[package.new]
  trigger package.new : At = the source
  response
    begin
      if package.new : Destination ≠
        ( a package.previous ||
          package.previous immediately < package.new
                              wrt PACKAGES_EVER_AT_SOURCE(*) ) : Destination
      then WAIT[] ;

      update : At of package.new to (the source) : Source_Outlet
    end ;

relation PACKAGES_EVER_AT_SOURCE(packages | sequence of PACKAGE)
  definition packages = ( a package ) ordered wrt package : At = ( the source ) ;
```

In English,

> *Determination of when to release a package into the network commences when a new package becomes located at the source station.*
> *If the new package's destination differs from that of the previous package (that is, the package immediately preceding the new package in the sequence of packages to have been located at the source (defined below)), then wait.*
>
> *The sequence of packages ever located at the source is defined by nondeterministically referring to packages, and ordering them by the order in which they were located at the source--derived relation*[3] **PACKAGES_EVER_AT_SOURCE**.

Historical reference proves to be of significant utility here because it frees the specifier of concerns about how to save information that is required but not readily available in the current computation state.

3.1.2 Mapping Away Historical References

Options for mapping historical references away are as follows.

- Introducing and maintaining auxiliary data structures to hold information which might be referenced, and modifying the historical references to extract the information from these introduced structures. The requirement for economy of storage in an implementation encourages the implementor to seek a compact representation for the information that need be preserved and to discard information once it is no longer useful
- Resolving the historical reference by derivation in terms of information available in the current state (without having to retain extra information from past states)

We suspect that the latter is rarely an available option. When it is, the two alternatives above present the classical space/time tradeoffs; we must still compare the cost of the derivation with the cost of storage and maintenance of redundant information to permit simple access. In the case of our example, the specification indicates no way to derive the identity of the previous package passing through the source station. Thus the first option, that of introducing auxiliary data structures, must be used in this case.

Two factors bear directly on the range of choice of how much, and what, information needs to be retained in an implementation of historical reference. These are the nature of the retrievals of the information and how it is used. Both these factors are evident in our chosen example.

First, through examination of the entire specification we see that the only occasion in which this particular historical reference is made is when a new package has entered the source station. Further, we refer only to the package immediately preceding the very latest package--earlier ones are no longer of any interest. Hence an implementation need remember only the most recent package to have passed through the source station rather than all of them.

Second, we see that the only information we require from the historically referenced previous package is its destination (in order to compare it with the destination of the new package). Since

---

[3] Another Gist construct, see Section 3.3.

destinations are static attributes of packages (a property easily provable of the specification), we could choose to remember only the preceding package's destination as opposed to its identity. In general we must compare the frequency of storing the information with the frequency of accessing it to determine whether it is better to perform the calculation upon storage or upon retrieval. In our simple example we can see that there will be more retrievals than stores (because it is necessary to store the destination only when it changes), so we choose to retain the destination rather than the identity. Thus our historical reference is mapped into the following:[4]

```
demon RELEASE_PACKAGE_INTO_NETWORK[package.new]
  trigger package.new : At = the source
  response
    begin
      if package.new : Destination ≠ PREVIOUS_PACKAGE_DESTINATION(*)
      then WAIT[] ;

      update : At of package.new to source : Source_Outlet;

      update ppdest in PREVIOUS_PACKAGE_DESTINATION($)
                    to package.new : Destination
    end ;

relation PREVIOUS_PACKAGE_DESTINATION(ppdest | BIN) ;
```

3.2 CONSTRAINTS AND NONDETERMINISM

Constraints within Gist provide a means of stating integrity conditions that must always remain satisfied. Within Gist, constraints are more than merely redundant checks that the specification always generates valid behaviors: constraints serve to *rule out* those behaviors that would be invalid. In conjunction with the use of nondeterminism, they serve as an extremely powerful specification mechanism, permitting us to describe an activity in a nondeterministic fashion; those behaviors of the activity leading to states that violate the constraint are "pruned away." We are thus able to express our intents more directly (in the form of constraints) rather than encoding all the processes of the specification so as to interact in only those ways that prevent arriving at an undesirable state.

3.2.1 Example From Package Router

```
always prohibited MORE_THAN_ONE_SOURCE
  exists source.1, source.2;
```

[4] Section 4 gives the actual development steps for reaching this state.

This constraint prohibits the existence of more than one source by defining a predicate that must never be true.[5]

In the package router, constraints such as MORE_THAN_ONE_SOURCE define the nature of the world in which the specified system will exist. The constraint does not directly constrain the part of the specification to be implemented. Constraints that do affect the behaviors of the portion to be implemented receive the most attention in the development of an implementation.

3.2.2 Another Example From Package Router

```
demon SET_SWITCH[switch]
  trigger Random()
  response
    begin
      require SWITCH_IS_EMPTY(switch);
      update : Switch_Setting of switch to switch : Switch_Outlet
    end;

always prohibited DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE
  exists package, switch ||
    ( PACKAGE_IN_CORRECT_SWITCH_WRONGLY_SET(package)
      and
      ( ( SWITCH_IS_EMPTY(switch) and
          package = first(PACKAGES_TO_BE_ROUTED_BY_SWITCH(*.switch) )
        ) asof true )
    );
```

The **SET_SWITCH** *demon is a nondeterministic expression of behaviors. It states that at random times the* **Switch_Setting** *of some nondeterministically selected switch should be set to a nondeterministically selected* **Switch_Outlet** *of that switch.* Picture a package router mechanism with switches flapping at random.

The system behavior we desire to specify is to route packages correctly whenever possible, given the limitation of not being able to change a switch's setting unless that switch is empty. The constraints that state this desired behavior are the require statement within the body of the SET_SWITCH demon and the DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE constraint. The former rules out those behaviors that involve setting a switch when it is not empty.[6] The latter defines a predicate that recognizes, after the fact, that a switch has not been correctly set. This recognition occurs when

[5] Variable names formed by appending distinct suffixes to a type name and "." (e.g., *source.1 source.2*) have the special meaning of denoting distinct objects.

[6] Since this is the unique place in the specification where the setting of switches is modeled, we have chosen to use a require statement rather than a global constraint (a stylistic choice).

- *a package is in a correct switch (that is, a switch on route to the package's destination) but the switch is set the wrong way, and*
- *at some time in the past there was a chance to set the switch for that package, that is, at some time*
  - *the switch was empty, and*
  - *the package was the first of those packages due to be routed by the switch (relation* PACKAGES_TO_BE_ROUTED_BY_SWITCH*).*

By putting this predicate into an always prohibited, behaviors which lead to such states are ruled out. Now picture a package router mechanism with switches flapping in just the right ways to get only desirable behaviors.

Constraints and historical reference have been used here to characterize and rule out those behaviors considered undesirable. We judge this to be a superior specification to one which uses a complicated encoding of the trigger of the switch setting demon to achieve the desired (and only the desired) behaviors.

3.2.3 Mapping Away Constraints and Nondeterminism

To fully appreciate the implementation freedoms afforded by the specification constructs of constraint and nondeterminism, it is necessary to understand the distinction between nondeterminism in the specification and nondeterminism in the implementation. It is obvious that in our example, the SET_SWITCH demon is a nondeterministic expression of behaviors. Its associated constraints serve to limit some aspects of this nondeterminism while completely eliminating others. For example, the DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE constraint demands that a switch be set correctly for the package next due to arrive *when that package finally arrives*. In this respect, the constraint prescribes fully deterministic behavior.

On the other hand, the constraint leaves imprecise (and therefore nondeterministic) the statement of when the switch must be set. In fact, the demon/constraint combination in our example admits behavior in which a switch flaps back and forth until the very last moment at which it can be correctly set. More precisely, the specification defines interval(s) during which the switch can be freely flapping, provided that at the end of the (last) interval the switch is set correctly for the package. Of course, an acceptable implementation could switch the switch just once (if necessary) to set it correctly for the next package due. In the implementation we derive for this specification, switches are set as early as possible for the simple reason that the beginning of the (first) interval is readily defined in terms of the available information about packages' positions, while the end of the (last) interval is not.

The most interesting case for consideration in deriving an implementation arises when specification nondeterminism and constraints combine to describe deterministic behavior (as in the case of determining the correct way to set a switch). A range of possibilities present themselves for mapping away such nondeterminism and constraints.

- The "predictive" solution. At one extreme, we might seek to introduce code into all the nondeterministic choice points to perform the necessary calculations. These calculations determine the choices that will not lead to disaster arbitrarily far into the future. In our example, this means adjusting SET_SWITCH to set switches at the right moments and in just the right ways.

- The "backtracking" solution. At the other extreme, we might derive a backtracking algorithm, with the choice points as the backtracking points and the constraints mapped into search-terminating checks at the appropriate places in the program. In our example, this means arbitrary setting of switches with backtracking when we discover that one of our constraints is violated, in which case we backtrack and try different settings. Unfortunately, we do not have the ability to move packages backwards through our network; this type of solution is therefore ruled out for the package router.[7]

The nature of the domain of the specification strongly influences the choice of method. The capabilities of the effectors[8] (if there are any in the system being specified), the amount of information available for making decisions, and the desired amount of precomputation all affect the choice of algorithm. In our example, for instance, there is no way to return a package to an incorrectly set switch in order to backtrack. However, static knowledge about the package router network topology is available in the specification, so it is possible to precompute the correct setting for each switch and destination bin. Typically, as in Balzer's Eight Queens problem, the interesting issue is to produce an algorithm to efficiently perform the search, not to make use of the result derived by the search (as in this example). We see a general preference for avoiding backtracking-type implementations if at all possible, and in our package router development we find a purely "predictive" solution.

Mapping away require **SWITCH_IS_EMPTY**(*switch*) within **SET_SWITCH** demon:

We simply add the predicate of the requirement as a conjunct to the demon's trigger, to get the following:

```
demon SET_SWITCH[switch]
  trigger Random() and SWITCH_IS_EMPTY(switch)
  response
    begin
      update : Switch_Setting of switch to switch : Switch_Outlet
    end;
```

It is easy to see that this mapping is valid, since the added conjunct rules out precisely those behaviors in which the **SET_SWITCH** demon would have triggered and violated the requirement.

To understand how we might map away the **DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE** constraint, we paraphrase it in order to simplify the discussion. Thus,

[7] See Balzer's Eight Queens development [1] for a solution of this nature.

[8] For example, switches

```
always prohibited DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE
  exists package, switch ||
    ( PACKAGE_IN_CORRECT_SWITCH_WRONGLY_SET(package)
      and
      ( ( SWITCH_IS_EMPTY(switch) and
          package = first(PACKAGES_TO_BE_ROUTED_BY_SWITCH(*.switch) ) )
                                                          asof true )
    ) ;
```

becomes

```
always prohibited DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE
  exists package, switch ||
    P(package) and ( Q(switch, package) asof true );
```

P corresponds to the presence of a package at a wrongly set switch, and Q to an opportunity to set the switch correctly for that package. The paraphrasing should be read as "avoid a state in which P holds and there has been a state (or states) in which Q held." We could seek to map this into a "predictive" solution by selecting those behaviors in which, whenever Q is about to hold (or holds, or will hold), we do something to ensure that P will not become true then or later. This corresponds to setting a switch correctly for a package whenever that package is the first of those due for the switch and the switch is empty.

Alternatively, we could select an implementation which assures that Q never holds. This implies that no package should ever become the first of those due at a switch when that switch is empty. It might be possible to accomplish this by conspiring to prevent there ever being an opportunity to set the switch correctly, say by always delaying the exit of a package from a switch until it is too late to switch for the next package. Note, however, that we do not have the power to control the speed at which packages flow through the network.

The issue concerning the methods by which an implementation preserves the integrity of constraints is one of the more difficult mapping problems. At the specification level, constraints apply to the whole specification. As implementors, however, we can control the behavior of only the portion of the specification we are implementing. We assume that our mission is to implement this portion in such a way as to ensure that such constraints are not violated independent of the behavior of other portions of the system, as long as they too meet their specifications. This means, for example, that we cannot expect (nor can we implement) the package moving mechanism to conveniently delay movement of all packages so as to avoid Q ever holding. *Some* requirements are imposed on the environment--for example, the constraint preventing packages from overtaking one another is entirely the responsibility of the environment--and we rely on these invariants rather than having to maintain them. But it is our responsibility to implement our portion to react to any allowable behavior of the

other portions of the specification in a way that will never allow any constraints to be violated.[9]

Finally, we arrive at the mapped version of the set switch demon, which utilizes "predictive" behavior to avoid producing a constraint violation:

```
demon SET_SWITCH[switch]
  trigger SWITCH_IS_EMPTY(switch) and
          exists package ||
              package = first(PACKAGES_TO_BE_ROUTED_BY_SWITCH(*,switch))
  response
    update : Switch_Setting of switch
        to (the pipe || ( pipe = switch : Switch_Outlet and
                    BELOW(package : Destination, pipe) ) );
```

3.3 DERIVED RELATIONS

One of the underlying features of Gist is that all knowledge is represented in terms of objects and relations[10] among those objects. Change is therefore represented by the creation or destruction of objects and by the insertion or deletion of relations among them.

Often it is convenient to make use of a relationship that is derived from other relationships. Its derivation is declared once and for all and serves to denote all the maintenance necessary to preserve the invariant between the derived relation and the relations upon which it depends.

3.3.1 Example From Package Router

```
relation PACKAGES_TO_BE_ROUTED_BY_SWITCH(packages | sequence of PACKAGE,
                                         switch)
  definition
    packages =
      ( a package ||
        ( BELOW(switch,package : At) and
          BELOW(package : Destination,switch) and
          not PACKAGE_IN_CORRECT_SWITCH_WRONGLY_SET(package) )
      ) ordered wrt PACKAGES_EVER_AT_SOURCE(*) ;
```

[9] Of course, such a one-sided division of responsibility may not be possible in a given situation (i.e., the specification is unimplementable). Avoiding constraint violation may require cooperation among the portions of the system. If so, the specification must be revised so that such cooperation is required from each portion and can therefore be relied upon.

[10] Attributes, such as Switch_Setting, are merely a syntactically convenient and commonly useful form of binary relation.

This relation defines **PACKAGES_TO_BE_ROUTED_BY_SWITCH** as a derived relation between switches and sequences of packages. The derivation is a predicate which relates the sequence of packages to the switch.

For any particular switch, the sequence of packages consists of those for which

- *the switch lies below the package's current location in the network,*
- *the package's destination lies below the switch, and*
- *the package is not in a correct switch that is wrongly set (which would imply that the package is doomed to be misrouted).*

The ordering puts packages in sequence by the time at which they were located at the source.[11]

3.3.2 Mapping Away Derived Relations

The specificational power of the derived relation construct comes from being able to state a derivation in a single place and to make use of the derived information throughout the specification. Since no corresponding construct is likely to be available in any implementation language we might choose,[12] we must map the derivation into explicit data structures and mechanisms to support all the uses of that information that are scattered throughout the program. We have a wide range of choices as to how we might do this mapping.

- At one extreme, we might simply unfold the derivation at all the places where a reference to the relation is made. Having done this, we may completely discard the relation and its derivation. For example, after performing the constraint removal in the previous section, we could unfold **PACKAGES_TO_BE_ROUTED_BY_SWITCH** in its sole place of use, within the trigger of the **SET_SWITCH** demon. (This relatively simple step would leave the bulk of the development effort to the mapping away of the demon construct.) This approach is analogous to backward inference, where computation is performed on demand and at the site of the need.
- At the other extreme, we might retain the relation but scatter throughout the program the code necessary to explicitly maintain the invariant between the derived information and the information upon which it depends ("base" information). Thus it is necessary to find all possible places where a change is made to any of the base information and to augment such places with code to recalculate the derived information. This approach is analogous to forward inference, where computation is performed whenever a modification to a relevant predicate occurs and at the site of the change.

---

[11] Observe that the structure of the network (a tree with the source at the root) and the property that packages cannot overtake one another combine to ensure that packages will arrive at switches in the same order in which they were located at the source.

[12] Many of the Artificial Intelligence programming languages do provide facilities for implementing derived relations in terms of inference processes. For example, an implementation of derived relations might be provided in CONNIVER [30] in terms of IF-ADDED or IF-NEEDED methods. However, AI programming languages in which these facilities are present typically do not provide for the efficient execution one would desire for an optimized implementation, nor do these facilities provide precisely the semantics desired without the inclusion of satisfactory "truth maintenance" capabilities [18, 28].

As with mapping away historical reference, the nature of the use of derived information affects our options and decisions. In our example, we might observe that only the first package of the sequence of packages due is ever required, and hence we might seek to maintain only that single piece of information rather than the entire sequence.

The choices among the implementation alternatives imply alternative trade-offs between storage and computation in the resulting program. Completely unfolding the derivation is tending towards complex recalculation with a minimum of stored data. Maintaining sequences of packages due at each switch is tending towards simplifying calculations by simple maintenance of additional data. Maintaining only the first package due at each switch is a compromise between these positions.

Thus, to explicitly maintain the relation **PACKAGES_TO_BE_ROUTED_BY_SWITCH**, it is necessary to determine the portions of the specification that cause changes to its "base" information:

1. Changes to the attributes **At** and **Destination** of packages, which occur when a new package is created at the source and when movement of a package occurs.
2. Changes to the derived relation **PACKAGE_IN_CORRECT_SWITCH_WRONGLY_SET**, which occur on changes to the attributes **At** and **Destination** of packages (as before) and on changes to the **Switch_Setting** attribute of switches.
3. Changes to the derived relation **PACKAGES_EVER_AT_SOURCE**, which occur when a new package is created at the source.

(Note that the derived relation **BELOW**, a property of the static structure of the routing network, is not changed by any activity in the specification.)

In performing a recalculation when base information changes, we may be able to take advantage of the state of knowledge prior to the change to incrementally update the derived information. For example, when a new package arrives at the source station, it can be appended to the end of each **PACKAGE_TO_BE_ROUTED_BY_SWITCH** sequence for those switches on the route from the source to that package's destination. Simply appending a new element onto the end of a sequence is easier than recalculating the entire sequence afresh. This is an example of a general technique we call "incremental maintenance," derived from the work of other researchers in set-theoretic settings, particularly Paige and Schwartz [33] and Paige and Koenig [32], who call the technique "formal differentiation," and Earley [19], who calls it "iterator inversion." Balzer's Eight Queens development includes the incremental maintenance of a set. Application of the incremental maintenance technique to **PACKAGES_TO_BE_ROUTED_BY_SWITCH** inserts code into the places identified above. Hence, **CREATE_PACKAGE** is transformed into the following:

```
demon CREATE_PACKAGE[]
  trigger Random()
  response
    atomic
      create package.new || package.new : Destination = a bin  and
            package.new : At = the source ;
      loop switch || BELOW(package.new : Destination, switch)
        do update packages
        of PACKAGES_TO_BE_ROUTED_BY_SWITCH(packages.switch)
        to packages concatenate package.new
    end atomic ;
```

In some places the introduced code can readily be eliminated because, although the base information is changing, the derived relation can be determined to always remain unchanged. This situation occurs in RELEASE_PACKAGE_INTO_NETWORK, since moving a package from the source into the first pipe does not cause the package to become located in a correct switch wrongly set.

Sometimes it is possible to relax the restriction that the invariant embodied by the relation definition need hold at all times. When maintaining a derived relation, if the uses of that relation do not immediately follow the changes to the base information, we may delay the incremental maintenance provided that it is performed before any information retrieval. Thus, the invariant only need hold at the times it is "used." To achieve this effect we may apply either sophisticated special-purpose transformations that deal with such delayed computation for maintenance or more standard ones that do "normal" incremental maintenance and then apply code-moving transformations to relocate the maintenance operations. The need for this may arise if incremental maintenance leads to inserting code into portions of the specification over which we have no control. This possibility is apparent in the above example, where we have altered the CREATE_PACKAGE demon, which is in fact part of the environment.

Hence, we must move the maintenance of PACKAGES_TO_BE_ROUTED_BY_SWITCH out of this demon. Our solution is to move the maintenance into the beginning of the response of the RELEASE_PACKAGE_INTO_NETWORK demon, which triggers on the arrival of a package at the source.

3.4 DEMONS

Demons are Gist's mechanism for providing data-directed invocation of processes. A demon's trigger is a predicate which triggers the demon's response whenever a state change induces a change in the value of the trigger predicate from false to true.

Demons are a convenient specification construct for use in situations in which we wish to trigger an activity upon some particular state change in the modeled environment. Demons save us from the need to identify the individual portions of the specification where actions might cause such a change

and the need to insert into such places the additional code necessary to invoke the response accordingly. The specificational power of the demon construct is enhanced by the power of Gist's other features, since the triggering predicate may make use of defined relationships, historical reference, etc.

3.4.1 Example From Package Router

```
demon RELEASE_PACKAGE_INTO_NETWORK[package.new]

  trigger package.new : At = the source

  response
    begin ... end ;
```

We saw this portion in Section 3.1.1 as an illustration of the use of historical reference. The trigger of this demon is a predicate that will become true whenever a package becomes located at the source. When the demon is so triggered, occurrences of the variable *package.new* in its response are bound to the instance of the object satisfying that triggering of the demon.

3.4.2 Mapping Away Demons

The demon RELEASE_PACKAGE_INTO_NETWORK is a special case insofar as its triggering condition is some external event, that is, the relevant state changes are produced only in the environment portions of the specification. The only mapping option is in how to implement the low-level details of detecting that event--for example, by using a polling loop or an interrupt-driven implementation. More interesting from the mapping point of view are the other demons in the portion of the specification to be implemented, SET_SWITCH and DETECT_MISROUTED_ARRIVAL, where the implementation must explicitly connect the production of the conditions upon which to trigger these demons with the invocation of the demons' response.

Mapping away such a demon involves identifying all places in the program where a state change might cause a change of the value of the demon's triggering predicate from false to true and inserting the code to make such a determination and perform the demon's response when necessary. This mapping has much in common with that for maintaining derived relations. Here, however, the information to be maintained is a truth value; its definition is the trigger predicate of the demon. Whereas in a derived relation a change would cause a maintenance operation of the information, here the demon's response is to be invoked.

Consideration of mapping away the SET_SWITCH demon dramatically illustrates the effect that the order in which constructs are eliminated may have on the development. If that demon were mapped away when its trigger was still in the form **Random()**, the result would be to augment *every* state change with a nondeterministic choice of whether or not to invoke that demon's response. Further manipulation of the resulting specification would be severely hampered, particularly in attempting the removal of constraints by limiting nondeterminism. Obviously it is far better to map away the

nondeterminism while it is localized in the demon (as discussed in Section 3.2) and then to map away the demon.

3.5 TOTAL INFORMATION

For specificational purposes it is convenient to make free use of any information about the world described by the specification. When implementing a specification, however, not all of the information is necessarily available at all times. The implementation specification describes what the portion to be implemented may observe and what it may effect.

3.5.1 Example From Package Router

```
implement PACKAGE_ROUTER
  observing
    types
      LOCATION, SOURCE, PIPE, SWITCH, BIN, PACKAGE;

    attributes
      Source_Outlet, Pipe_Outlet, Switch_Outlet, Switch_Setting;

    predicates
      start ( a package ) : At = the source,
      start ( a package ) : At = switch,
      finish ( a package ) : At = switch,
      start ( a package ) : At = bin;
      exists package || ( package : Destination = bin and package : At = the source );

  effecting
    attributes
      switch : Switch_Setting,
      package : At asof package : At = the source;
end implement
```

At several places within the specification it is assumed that the location and destination of packages anywhere within the network may be determined (uses of **At** and **Destination**). Examination of the implementation specification reveals that the only available information about these attributes of packages is the arrival of a package at the source, the movement of a package into or out of a switch or into a bin, and the destination of a package at the time it is located at the source.

3.5.2 Mapping Away Reliance on Total Information

Mapping away this reliance is similar to mapping away historical reference--either by introducing and maintaining auxiliary data structures to hold information that we might need to reference or by finding ways to derive the required information from other, available information.

Mapping away references to the **Destination** attribute of packages might be achieved by reading each package's destination upon its arrival at the source, explicitly remembering that value in the implementation, and adjusting retrievals to the destination attribute into retrievals from the remembered value (valid because packages' destinations remain static).
We could achieve the same result in two stages by first mapping in uses of historical reference, replacing retrievals of the form *package* : **Destination** with

(*package* : **Destination**) asof *package* : **At** = (the *source*)

and then mapping away these historical references by explicitly remembering the information.

Mapping away references to the **At** attribute of packages could be performed in a similar fashion, by remembering sequences of packages at locations within the network. Maintenance of these sequences would involve making use of sensor triggerings to indicate that packages have moved into or out of a location and updating the remembered information accordingly. In some cases the lack of perfect knowledge may constrain our choice of implementation by forcing us to select only those implementations guaranteed to make no reliance upon information that cannot be derived from the observable behavior.

As with mapping away other constructs, the context in which unobservable information is used and the nature of its use may affect our decisions about how to do the mapping. Quite often there are potential interactions between how we do this and other kinds of mappings; for example, consider the mapping away of the derived relation **PACKAGES_TO_BE_ROUTED_BY_SWITCH** (Section 3.3.2). Since we now see that we might need to maintain sequences of packages at each location in the specification, we might wish to choose a mapping of the derived relation which finds the first package due at a switch by looking through the sequences of packages at locations above that switch to find a correctly routed package. Thus, had we chosen some other mapping of the derived relation, we might now be motivated to go back and modify that choice accordingly.

4. DEVELOPMENT

In the previous section, we described some of the high-level constructs of Gist and discussed the freedoms these constructs provide in both specification (by easing the task of formally stating requirements) and implementation (by being neutral towards alternate implementations). Our emphasis has been more on the implementation aspects of these high-level constructs, and we have enumerated several alternative implementations for each. These mappings are predicated on the existence and applicability of equivalence-preserving transformations to effect the desired changes to the specification. In this section, we argue that the mappings described previously *can* be achieved by applications of relatively straightforward equivalence-preserving transformations and invocations of general-purpose mechanisms (such as incremental maintenance). To this end, we present an annotated development which derives the mapping for historical reference, described in Section 3.1.

Recall that we wish to eliminate historical references in the following portion of the package router specification:

```
demon RELEASE_PACKAGE_INTO_NETWORK[package.new]
  trigger package.new : At = the source
  response
    begin
      if package.new : Destination ≠
        ( a package.previous ||
          package.previous immediately < package.new
                              wrt PACKAGES_EVER_AT_SOURCE(*) ) : Destination
      then WAIT[] ;

      update : At of package.new to (the source) : Source_Outlet
    end ;

relation PACKAGES_EVER_AT_SOURCE(packages | sequence of PACKAGE)
  definition packages = ( a package ) ordered wrt package : At = ( the source ) ;
```

We desire to produce the following mapped version of the specification:

```
demon RELEASE_PACKAGE_INTO_NETWORK[package.new]
  trigger package.new : At = the source
  response
    begin
      if package.new : Destination ≠
        PREVIOUS_PACKAGE_DESTINATION(*)
      then WAIT[] ;

      update : At of package.new to (the source) : Source_Outlet;

      update ppdest in PREVIOUS_PACKAGE_DESTINATION($)
        to package.new : Destination
    end ;

relation PREVIOUS_PACKAGE_DESTINATION(ppdest | BIN);
```

The process of discovery of the sequence of transformations to effect this change is one of interaction between implementor and system. The implementor provides guidance by selecting transformations to be applied and the system applies them, maintaining the specification. We strongly expect the process to be an exploratory one, involving exploration of several alternatives, retracing steps, reordering steps, and reapplying transformations. Hence it is important that the system provide an environment conducive to such exploration.

In the development that follows we present a sequence of steps that lead directly towards our goal. We had to perform some exploration ourselves to discover this sequence; however, since our aim is to demonstrate that such a sequence exists rather than to show the process of discovery, we do not attempt to show any of the preceeding discovery activities.

In what follows, the result of each development step is displayed. The symbol ▸ at the side of a portion indicates an introduction or modification.

4.1 THE DEVELOPMENT

The development's first few steps take note of the context of use of the historically defined sequence **PACKAGES_EVER_AT_SOURCE** within the **RELEASE_PACKAGE_INTO_NETWORK** demon. In this context, *package.previous* is defined to be the package that precedes *package.new* in the sequence. One approach to eliminating the historical reference would involve explicitly maintaining the **PACKAGES_EVER_AT_SOURCE** sequence. Then it would be necessary to search that sequence for *package.new* and to return *package.previous* as the immediately preceding package. However, a more efficient implementation, which avoids maintaining the entire sequence, can be derived by noting the explicit relationship between *package.new* and the **PACKAGES_EVER_AT_SOURCE** sequence; *package.new* is precisely the last element. Thus, *package.previous* can be defined more directly in terms of this explicit definition of *package.new*.

Our goal in the development is maintenance of *package.previous*. However, because the definition of *package.previous* depends on **PACKAGES_EVER_AT_SOURCE**, we must first maintain **PACKAGES_EVER_AT_SOURCE**.

Step 1

Should we desire to re-express some expression in terms of the sequence we are about to maintain, we should perform such re-expression prior to the maintenance (since maintenance will replace the compact definition of the sequence with a less transparent algorithm for maintaining it). Thus the first step in this development involves the implementor making the observation that *package.new* (in the definition of *package.previous* in **RELEASE_PACKAGE_INTO_NETWORK**) is equivalent to last(**PACKAGES_EVER_AT_SOURCE**(*)). The transformation step is to replace that occurrence of *package.new* with its equivalent expression. The equivalence itself may be regarded as a lemma requiring verification. We expect that the implementor will suggest automated assistance for proving lemmas of this nature.[13]

The following shows the resulting specification after this transformation step:

```
demon RELEASE_PACKAGE_INTO_NETWORK[package.new]
  trigger package new : At = the source
  response
    begin
      if package.new : Destination ≠
        ( a package previous ||
          package previous immediately < last(PACKAGES_EVER_AT_SOURCE(*))
                          wrt PACKAGES_EVER_AT_SOURCE(*) ) : Destination
      then WAIT[] :

      update : At of package.new to (the source) : Source_Outlet
    end :

relation PACKAGES_EVER_AT_SOURCE(packages | sequence of PACKAGE)
  definition packages = ( a package ) ordered wrt package : At = ( the source ) :
```

Step 2

A standard method for maintaining a sequence such as **PACKAGES_EVER_AT_SOURCE** requires the introduction of a demon to add new elements when appropriate and a demon to remove elements when appropriate.[14] In this case, we use a somewhat specialized sequence-maintenance transformation designed for sequences with the form

(a *x*) ordered wrt P(*x*)

[13] Alternatively, this modification might have been suggested by a symbolic evaluation of this portion of the specification. Thus, the initiative for suggesting the validity of this step might have come from the system rather than from the implementor. Currently, the transformation application system we use [38] has neither a theorem prover nor a symbolic evaluator for Gist, both are in the design stage.

[14] This corresponds to the second mapping option from Section 3.3.2. However, rather than "scattering code," we use a demon which can be thought of as a procedure which has implicit calls scattered about the program.

We will drop the intermediate development steps that simplify the result of this transformation application by eliminating the demon that removes elements when, as here, elements of that type are never destroyed.

Two remarks need be made concerning this development step:

1. If PACKAGES_EVER_AT_SOURCE is used elsewhere in the specification, then changes we make to it here should be made to a local copy (i.e., a new relation with a different name but an identical definition).
2. The introduced demon, NOTICE_NEW_PACKAGE_AT_SOURCE, must execute before demon RELEASE_PACKAGE_INTO_NETWORK, which has an identical trigger.

```
demon RELEASE_PACKAGE_INTO_NETWORK[package.new]
  trigger package.new : At = the source
  response
    begin
      if package.new : Destination ≠
        ( a package.previous ||
          package.previous immediately < last(PACKAGES_EVER_AT_SOURCE(*))
                              wrt PACKAGES_EVER_AT_SOURCE(*) ) : Destination
      then WAIT[] :

      update : At of package.new to (the source) : Source_Outlet
    end :

relation PACKAGES_EVER_AT_SOURCE(packages | sequence of PACKAGE) :

demon NOTICE_NEW_PACKAGE_AT_SOURCE[package.new]
 trigger package.new : At = the source
 response update packages in PACKAGES_EVER_AT_SOURCE(packages)
          to packages concatenate package.new ;
```

The effect of the above transformation has been to introduce the **NOTICE_NEW_PACKAGE_AT_SOURCE** demon. This demon triggers whenever a package becomes located at the source, and it responds by updating the **packages** in **PACKAGES_EVER_AT_SOURCE** to be the old value of the **packages** sequence concatenated with the new package.

Note that at this stage the historical references we were concerned with have been eradicated; however, we are still some distance from the form of the program we seek, and the remaining steps cover that distance.

Step 3

In this step, the implementor isolates the portion defining *package.previous* by extracting the definition of *package.previous* from the demon and defining a new unary relation which is true of only the penultimate element of PACKAGES_EVER_AT_SOURCE(*). This definition is accomplished by application of a standard *fold into relation definition* transformation. One form of fold transformation takes an object expression and replaces it with a retrieval from a relation defined as that object expression. Application of this transformation yields the following:

```
demon RELEASE_PACKAGE_INTO_NETWORK[package new]
  trigger package.new : At = the source
  response
    begin
      if package.new : Destination ≠
        PREVIOUS_PACKAGE(*) : Destination
      then WAIT[] ;

      update : At of package.new to (the source) : Source_Outlet
    end ;

relation PACKAGES_EVER_AT_SOURCE(packages | sequence of PACKAGE) :

relation PREVIOUS_PACKAGE(prev_package | PACKAGE)
  definition
    prev_package = (the package.previous ||
              package.previous immediately < last(PACKAGES_EVER_AT_SOURCE(*))
                   wrt PACKAGES_EVER_AT_SOURCE(*) ) :

demon NOTICE_NEW_PACKAGE_AT_SOURCE[package.new]
  trigger package.new : At = the source
  response update packages in PACKAGES_EVER_AT_SOURCE(packages)
         to packages concatenate package.new ;
```

Step 4

To accomplish the next step, maintaining PREVIOUS_PACKAGE itself, we again exercise the second mapping option from Section 3.3.2.

```
demon RELEASE_PACKAGE_INTO_NETWORK[package.new]
  trigger package.new : At = the source
  response
    begin
      if package.new : Destination ≠ PREVIOUS_PACKAGE(*) : Destination
      then WAIT[] ;

      update : At of package.new to (the source) : Source_Outlet
    end ;

relation PACKAGES_EVER_AT_SOURCE(packages | sequence of PACKAGE) :

▸ relation PREVIOUS_PACKAGE(prev_package | PACKAGE) ;

demon NOTICE_NEW_PACKAGE_AT_SOURCE[package.new]
  trigger package.new : At = the source
  response
    atomic
      update packages in PACKAGES_EVER_AT_SOURCE(packages)
          to packages concatenate package.new ;
▸     update prev_package in PREVIOUS_PACKAGE($)
▸         to ( the package.previous ||
▸               package.previous immediately <
▸               last(PACKAGES_EVER_AT_SOURCE(*) concatenate package.new)
▸                  wrt PACKAGES_EVER_AT_SOURCE(*) concatenate package.new)
    end atomic ;
```

Code for the purpose of maintaining PREVIOUS_PACKAGE has been introduced into the NOTICE_NEW_PACKAGE_AT_SOURCE demon at the point where a modification is made to the relation PACKAGES_EVER_AT_SOURCE, upon which PREVIOUS_PACKAGE depends. The effect of this transformation is to *unfold* the definition of PREVIOUS_PACKAGE to the possible points of change (in this case, only one). That is, after every possible change to relevant data, code is introduced to perform an entire recalculation, determining if there was actually a change to PREVIOUS_PACKAGE.

The code to update the value of PREVIOUS_PACKAGE lies inside a newly created atomic, which is a Gist construct that allows a "macro" state change. That is, several data base modifications can be made inside an atomic, within which all changes occur with respect to the state preceding the atomic and no state transition occurs until the atomic is completed. Thus, since the updated value of PREVIOUS_PACKAGE wishes to refer to the updated value of PACKAGES_EVER_AT_SOURCE, it must reproduce the modification to PACKAGES_EVER_AT_SOURCE because the modification is not reflected inside the atomic. The update of PREVIOUS_PACKAGE is to "the package immediately preceding the last package in the sequence consisting of the old value of PACKAGES_EVER_AT_SOURCE concatenated with the new package."

Step 5

Having achieved the maintenance of PREVIOUS_PACKAGE, we now perform a local simplification on the code that was inserted to do that maintenance. The transformation

```
(the obj.previous || obj.previous immediately < obj.last
                                  wrt (sequence concatenate <obj.last>))
-->
(the obj.previous || obj.previous = last(sequence))
```

is applied to simplify the expression of the value to which *prev_package* is updated. This transformation is clearly derivable from the properties of concatenate and immediately <. It can be expressed as a chain of simpler transformations dealing with these constructs. This transformation can be verified in several ways. If it is not listed in our catalogue of already verified transformations, we can describe the change ourselves and require either that its correctness be verified or that the equivalent chain of simpler transformations be discovered. (The latter technique is under investigation by another member of our group [23]).

The result of the transformation (however verified) is as follows:

```
demon RELEASE_PACKAGE_INTO_NETWORK[package.new]
  trigger package.new : At = the source
  response
    begin
      if package.new : Destination ≠ PREVIOUS_PACKAGE(*) : Destination
      then WAIT[] ;

      update : At of package.new to (the source) : Source_Outlet
    end ;

relation PACKAGES_EVER_AT_SOURCE(packages | sequence of PACKAGE) ;

relation PREVIOUS_PACKAGE(prev_package | PACKAGE) ;

demon NOTICE_NEW_PACKAGE_AT_SOURCE[package.new]
  trigger package.new : At = the source
  response
    atomic
      update packages in PACKAGES_EVER_AT_SOURCE(packages
          to packages concatenate package.new ;
      update prev_package in PREVIOUS_PACKAGE($)
          to ( the package.previous ||
                  package.previous =
                  last( PACKAGES_EVER_AT_SOURCE(*) )
    end atomic ;
```

Step 6

This step applies a trivial local simplification to the code resulting from Step 5. We intend that our transformation system perform such simplifications automatically. The simplification is

(the *x* || *x* = EXPR) --> EXPR

This transformation requires that the type of EXPR be equal to or a subtype of *x*, and that EXPR be a deterministic expression. These properties are easily proved, the former because Gist is strongly typed, the latter because last of a deterministic sequence must be deterministic, and PACKAGES_EVER_AT_SOURCE(*) can be shown to be a deterministic sequence.

The resulting code is as follows:

```
demon RELEASE_PACKAGE_INTO_NETWORK[package.new]
  trigger package.new : At = the source
  response
    begin
      if package.new : Destination ≠ PREVIOUS_PACKAGE(*) : Destination
      then WAIT[] ;

      update : At of package.new to (the source) : Source_Outlet
    end ;

relation PACKAGES_EVER_AT_SOURCE(packages | sequence of PACKAGE) :

relation PREVIOUS_PACKAGE(prev_package | PACKAGE) ;

demon NOTICE_NEW_PACKAGE_AT_SOURCE[package.new]
 trigger package.new : At = the source
 response
   atomic
     update packages in PACKAGES_EVER_AT_SOURCE(packages)
          to packages concatenate package.new ;
     update prev_package in PREVIOUS_PACKAGE($)
          to last( PACKAGES_EVER_AT_SOURCE(*) )
   end atomic ;
```

Step 6 concludes the second sequence of steps. At this point we once again take stoc of the situation and plan a few "cleanup" steps. An observation that motivates the cleanup is that the explicit maintenance of PREVIOUS_PACKAGE involves the expression

last(PACKAGES_EVER_AT_SOURCE(*))

Our cleanup will be to fold this expression into a new relation, explicitly maintain that relation, and simplify--Steps 7 through 9 (akin to the sequence of steps we have just completed).

Step 7

As in Step 2, we isolate the interesting portion from the rest of the specification by extracting it and using it to create the definition of a new relation: last(PACKAGES_EVER_AT_SOURCE(*)) becomes the definition of the new relation **LAST_PACKAGE**.

```
demon RELEASE_PACKAGE_INTO_NETWORK[package.new]
  trigger package.new : At = the source
  response
    begin
      if package.new : Destination ≠ PREVIOUS_PACKAGE(*) : Destination
      then WAIT[] ;

      update : At of package.new to (the source) : Source_Outlet
    end ;

relation PACKAGES_EVER_AT_SOURCE(packages | sequence of PACKAGE) ;

relation PREVIOUS_PACKAGE(prev_package | PACKAGE) ;

demon NOTICE_NEW_PACKAGE_AT_SOURCE[package.new]
  trigger package.new : At = the source
  response
    atomic
      update packages in PACKAGES_EVER_AT_SOURCE(packages)
          to packages concatenate package.new ;
      update prev_package in PREVIOUS_PACKAGE($)
          to LAST_PACKAGE(*)
    end atomic ;

relation LAST_PACKAGE(last_package | PACKAGE)
definition
  last_package = last( PACKAGES_EVER_AT_SOURCE(*) ) ;
```

Step 8

Maintaining **LAST_PACKAGE** involves mapping a derived relation. We will again exercise the second mapping option from Section 3.3.2.

```
demon RELEASE_PACKAGE_INTO_NETWORK[package.new]
  trigger package.new : At = the source
  response
    begin
      if package.new : Destination ≠ PREVIOUS_PACKAGE(*) : Destination
      then WAIT[] ;

      update : At of package.new to (the source) : Source_Outlet
    end ;

relation PACKAGES_EVER_AT_SOURCE(packages | sequence of PACKAGE) ;

relation PREVIOUS_PACKAGE(prev_package | PACKAGE) ;

demon NOTICE_NEW_PACKAGE_AT_SOURCE[package.new]
  trigger package.new : At = the source
  response
    atomic
      update packages in PACKAGES_EVER_AT_SOURCE(packages)
          to packages concatenate package.new ;
      update prev_package in PREVIOUS_PACKAGE($)
          to LAST_PACKAGE(*) ;
      update last_package in LAST_PACKAGE($)
          to last( PACKAGES_EVER_AT_SOURCE(*) concatenate < package.new > )
    end atomic ;

relation LAST_PACKAGE(last_package | PACKAGE) ;
```

We have invoked precisely the same transformation as in Step 4, where we maintained PREVIOUS_PACKAGE. Again, the maintenance of LAST_PACKAGE occurs inside the atomic of NOTICE_NEW_PACKAGE_AT_SOURCE and is thus defined in terms of the old value of PACKAGES_EVER_AT_SOURCE concatenated with the new package.

Step 9

This step again invokes a trivial local simplification of the form

last(*sequence* concatenate <*element*>) --> *element*

```
demon RELEASE_PACKAGE_INTO_NETWORK[package.new]
 trigger package.new : At = the source
 response
   begin
     if package.new : Destination ≠ PREVIOUS_PACKAGE(*) : Destination
     then WAIT[] ;

     update · At of package.new to (the source) : Source_Outlet
   end ;

relation PACKAGES_EVER_AT_SOURCE(packages | sequence of PACKAGE) ;

relation PREVIOUS_PACKAGE(prev_package | PACKAGE) ;

demon NOTICE_NEW_PACKAGE_AT_SOURCE[package.new]
 trigger package.new : At = the source
 response
   atomic
     update packages in PACKAGES_EVER_AT_SOURCE(packages)
          to packages concatenate package.new ;
     update prev_package in PREVIOUS_PACKAGE($)
          to LAST_PACKAGE(*) ;
     update last_package in LAST_PACKAGE($)
          to package.new
   end atomic ;

relation LAST_PACKAGE(last_package | PACKAGE) ;
```

Step 9 completes the third small sequence of transformation steps, and we appear to have achieved as much local simplification as possible. Next we will make a simplification based upon a more global observation.

Step 10

By observing (and having the system prove) that PACKAGES_EVER_AT_SOURCE is maintained but never referenced, we can remove the relation entirely, with the following result:

```
demon RELEASE_PACKAGE_INTO_NETWORK[package.new]
  trigger package.new : At = the source
  response
    begin
      if package.new : Destination ≠ PREVIOUS_PACKAGE(*) : Destination
      then WAIT[] ;

      update : At of package.new to (the source) : Source_Outlet
    end ;

relation PREVIOUS_PACKAGE(prev_package | PACKAGE) ;

demon NOTICE_NEW_PACKAGE_AT_SOURCE[package.new]
  trigger package.new : At = the source
  response
    atomic
      update prev_package in PREVIOUS_PACKAGE($)
          to LAST_PACKAGE(*) ;
      update last_package in LAST_PACKAGE($)
          to package.new
    end atomic ;

relation LAST_PACKAGE(last_package | PACKAGE) ;
```

4.2 DISCUSSION

4.2.1 Summary of Development Steps

Maintain PACKAGES_EVER_AT_SOURCE (Step 2); but prior to doing so, re-express *package.previous* in terms of PACKAGES_EVER_AT_SOURCE (Step 1).

Isolate the expression for *package.previous* (Step 3) and maintain it (Step 4); then perform local simplifications (Steps 5 and 6).

Isolate the expression for last(PACKAGES_EVER_AT_SOURCE(*)) (Step 7), maintain it (Step 8), and then perform local simplification (Step 9) and global simplification (Step 10).

4.2.2 Remaining Minor Steps

The major objective of this part of the development, namely, the elimination of historical reference in a respectably efficient manner, has been attained. The implementation is not yet in the precise form of the stated goal of this development; the steps to achieve this form are similar to those already presented. They involve noticing that

1. the **NOTICE_NEW_PACKAGE_AT_SOURCE** demon has the same trigger as **RELEASE_PACKAGE_INTO_NETWORK**. Thus their responses can be combined (taking care that proper order is preserved).
2. only the **Destination** attribute of the previous package is desired: thus it is not necessary to save the identity of the package, only its destination.
3. because **LAST_PACKAGE** equals *package.new* inside the response of the **RELEASE_PACKAGE_INTO_NETWORK** demon, we do not need to explictly remember **LAST_PACKAGE** (or its destination).

4.2.3 Summary

The objective of this section was to demonstrate that high-level mappings can be accomplished through judicious application of low-level transformations. The other mappings described in Section 3 are similarly achievable.

5. RELATED WORK

5.1 TRANSFORMATIONAL METHODOLOGY

The transformational methodology that we follow is one of several approaches to improving software development. The research we have described relates most closely to those efforts involving some form of mapping between constructs on different levels of programming. In this section, we comment on several such efforts.

Burstall and Darlington. Their early schema-based transformations [12, 16] served to do optimization tasks of recursion removal, common subexpression elimination, loop combination, in-line procedure expansion, and introduction of destructive operations to reuse released storage. These techniques were built into a system which allowed the user to select the optimization process to be attempted next. Burstall and Darlington made the observation that manipulations are better done on higher level programs before mappings down to the next level (recursion to iteration, procedure expansion, etc.) are performed.
Their later work concentrated upon manipulation of recursion equations and achieved efficiency improvements by altering the recursion structure [10]. The changes were primarily radical restructuring of algorithms, achieving efficiency at the expense of modularity and clarity. These ideas were embodied in an experimental system that relied upon a semiautomatic approach. The system proposed possible steps to take, and the user selected or rejected avenues for exploration [15]. Another system based on the same ideas was developed to tackle larger scale examples by requiring (and permitting) the user to provide guidance in a more high-level fashion [20, 22]. Darlington extended the underlying approach to be able to go from specifications initially containing set and sequence constructs into recursion equations [13, 14]. Darlington's research in these and related directions continues.
We see many issues crucial to the overall transformational methodology being pursued in this research. The most significant difference (from our own efforts) lies in the nature of the specification language. Their language, HOPE [11] (formerly NPL), is purely applicative in nature; although they are able to investigate many of the issues of transformational development of software within this applicative framework, we believe that an applicative specification language is limited in the nature of the problems to which it is suited. In contrast, our language, Gist, has been constructed explicitly to express a wide range of tasks.

Manna and Waldinger. Their DEDALUS system [29] comprised a fully automated approach to deriving algorithms in some target language from specifications in some specification language rich with constructs from the subject domain of the application. The examples they have dealt with involve specifications using set theoretic constructs; these specifications become synthesised into recursive procedures and in turn into iterative procedures (in a LISP-like language). In their investigations the scale of the problems has been rather small because of their desire to do the synthesis in a fully automatic fashion. In contrast, the emphasis of our research has been to investigate tools which will assist a skilled developer to derive implementations from specifications. We hope that as we gain more experience with this activity we will incrementally introduce more automation into our tools.

CIP. The CIP (Computer-aided Intuition-guided Programming) group at Munich [5, 6] advocates using machine support to do the bookkeeping tasks in development and documentation, with a human user providing the intuition to guide this process. Their specification language is built upon a

kernel of recursive functions together with transformations to manipulate the kernel. Language extensions can be defined in terms of the kernel by application of the transformations. Thus new constructs can be defined for both implementation (e.g., loops, assignments, blocks) and specification (e.g., "descriptive expressions" corresponding to Gist's (the x || P(x)), abstract data types, nondeterministic expressions which could denote zero, one or more objects) [7]. In order to transform programs making use of these introduced constructs, the defining transformations are used to convert the constructs into recursive procedures (where substantial efficiency improvements may be achieved by applying the kernel transformations) before mapping them back into the desired constructs. As with the Burstall and Darlington work, there is much overlap between this work and our own in the approach and the research avenues being explored.

SETL. This work is based on the idea of augmenting specifications with guidance to a sophisticated compiler to suggest data representations [17, 35]. SETL is based around liberal use of tuples, sets, and functions (and operations upon them). User-provided declarations direct the compiler to select appropriate data structures (from a pre-existing library). The sophisticated compiler automatically generates the code to implement the operations on these representations. The group continues to investigate the usefulness of this approach and the extent to which currently user-made decisions can be automated [34]. As mentioned in Section 3, we are able to incorporate into our framework their techniques for dealing with some "data" freedoms, which should save us from the "reinventing the wheel" syndrome.

Neighbors. He advocates a methodology based on picking some domain of tasks, developing a set of reusable components for that domain, and then, when faced with a specific problem in that domain, combining and tailoring those components for that problem [31]. While this differs from the methodology we follow, parallels can be drawn between some of the details of, and observations drawn from, each approach. Neighbors' "refinements," which convert "components" (objects or operations) from one domain into another domain closer to implementation, correspond to our mappings to eliminate Gist constructs. Although the constructs expressed in the components of domains he has so far considered are not as rich as those within Gist, he has been led to similar observations, for example, that optimizations (his "customising" by transformation) are best done at the appropriate levels, that retaining the refinement record is helpful to maintenance, and that the choice of "refinement" (mapping to us) for one component may influence and interact with the refinement of other components.

5.2 SPECIFICATION

We believe that the approach to constructing our specification language, and the resulting combination of features, is new. However, analogies of individual features can be found in other languages.

- The relational data base model espoused by Smith and Smith [37].
- Temporal logic, at least in its use to talk and reason about the past. Gist's use of historical reference is very close to the approach of Sernadas in his temporal process specification language, DMTLT [36].
- Automatic demon invocation, seen in the AI languages PLANNER and Qlisp [9].
- Nondeterminism in conjunction with constraints--closest to nondeterministic automata theory [27].

- Operational semantics and closed system assumptions--as seen in simulation languages [8]. Zave's executable specification language PAISLEY [40, 41].

5.3 GROUP EFFORTS AT ISI

Below is an outline of the efforts that our group at ISI has performed. References to these subjects occur earlier in this report; we gather them together here to clarify the relation of the mapping component to the whole.

- Methodology. An outline of our overall approach is given in [3]. A detailed case study of a single development (the Eight Queens problem) is presented in [1].
- Specification language. Some requirements of a specification language suitable for system specification can be found in [2], and [25] provides a description of Gist, the language we are developing to satisfy those requirements.
- Supporting system for development. The POPART system, which produces tools to support our development process, is described in [39], and [38] contains a detailed discussion of issues relating to making the development process itself a formal object. A mechanism for automatically producing ("jittering") the many mundane steps that occur in a development is discussed in [23]. Research towards automating the higer levels of transformational development is presented in [24].
- Construction of specifications. Research aimed at supporting the construction of formal specifications from informal natural language expositions is reported in [4].

6. CONCLUSIONS

The primary goal of the research described in this paper has been the investigation of how to map away uses of Gist's high-level specificational constructs. An example specification provides the underlying motivation for, and an illustration of, our efforts. However, we have considered the task in more general terms; in so doing, we have provided a further illustration of the utility of Gist's constructs for specification.

The usefulness of our efforts can only be judged within the larger framework of the Transformational Implementation approach to software development. As a part of this overall methodology for developing software, our efforts depend on the success of that methodology. It remains to be demonstrated that Gist specifications can be readily constructed and manipulated and that the result of our mappings (a Gist specification with all uses of its high-level constructs mapped away) can readily be converted into an efficient program in a conventional implementation language. Our group is involved in continuing research in these areas.

We have described techniques for mapping away each of Gist's high-level constructs, suggested criteria for selecting one technique over another in terms of the specification and the desired nature of the implementation, and demonstrated that the mappings are derivable by the application of low-level equivalence-preserving transformations. However, some major tasks remain. Important among these are the tasks of actually compiling a sizable library of transformations for use in mapping activities and developing mechanisms to assist an implementor to carry out the mappings. We are now confident, however, that it will indeed be possible to both describe and collect such transformations and to make use of them in actual developments.

While pursuing the research described in this paper, we confronted several issues that bear further investigation.

- The order in which constructs are mapped away seems to be important. We do not expect there to be a "best" order independent of the problem. There seems to be an opportunistic component of transformational program development.
- Although there might be standard mappings to convert uses of one construct into uses of another, we do not think that a viable approach could be based on converting uses of all types of constructs into uses of just one type and then concentrating on mapping away that construct.
- The need to map two separate constructs occurring in disparate sections of the specification may lead to sharing of data structures or procedures. Thus the selections of mappings cannot be made independently; each might derive an optimal implementation for its instance, yet together they provide a suboptimal implementation for both. It is unlikely that we will be able to forsee all the ramifications of mapping decisions. Hence, we may expect to cycle back through the development process to adjust some of our earlier choices. This further highlights the need for machine support during the development process. With such support, exploratory development should be a relatively painless activity.
- Dealing with the distinction between system (the portion of the specification to be implemented) and environment (the remaining portions of the specification which establish the framework within which the system will operate) is very difficult. Often, mappings distribute code not only through the system portion of the specification, but also through the description of the environment, thus modifying its behavior. This

indicates that the implementation chosen requires cooperation from the environment. Since such cooperation cannot be assured (because it was not part of the specification), either another implementation must be chosen or the specification must be renegotiated (as is often necessary when implementation problems arise).

I. GIST SPECIFICATION OF PACKAGE ROUTER

Key to font conventions and special symbols used in Gist

symbol	meaning	example
|	of type	*obj* | T - object *obj* of type T
||	such that	(an *integer* || (*integer* > 3)) - an integer greater than 3
_	may be used to build names	this_name
.	concatenates a type name with a suffix to form a variable name, with the semantics that such variables with distinct suffixes denote distinct objects.	*integer.1*

fonts	meaning	example
underlined	key word	begin. definition. if
SMALL CAPITALS	type name	INTEGER
lower case italics	variable	*x*
UPPER CASE BOLDFACE	action. demon. relation, and constraint names	SET_SWITCH
Mixed Case Boldface	attribute names	Destination

Package Router Specification in Gist

The network

type LOCATION() supertype of

< SOURCE(Source_Outlet | PIPE);

Gist comment - the above line defines SOURCE to be a type with one attribute, Source_Outlet, and only objects of type PIPE may serve as such attributes. end comment

PIPE(Pipe_Outlet | (SWITCH union BIN) ::unique);

SWITCH(Switch_Outlet | PIPE :2, Switch_Setting | PIPE)
where always required
switch : Switch_Setting = *switch* : Switch_Outlet end;

BIN()
>;

Spec comment - of the above types and attributes, only the SWITCH_SETTING attribute of SWITCH is dynamic in this specification; the others remain fixed throughout. end comment

Gist comment - by default, attributes (e.g., Source_Outlet) of types (e.g., SOURCE) are functional (e.g., there is one and only one pipe serving as the Source_Outlet attribute of the SOURCE). The default may be overridden, as

occurs in the **Switch_Outlet** attribute of SWITCH; there, the ":2" indicates that each switch has exactly two pipes serving as its **Switch_Outlet** attribute *end comment*

```
always prohibited MORE_THAN_ONE_SOURCE
  exists source.1, source.2;
```

Gist comment constraints may be stated as predicates following either *always required (in which case the* predicate must always evaluate to true) or always prohibited (in which case the predicate must never evaluate to true). The usual logical connectives, quantification, etc., may be used in Gist predicates. Distinct suffixes on type names after exists have the special meaning of denoting distinct objects. end comment

```
always required PIPE_EMERGES_FROM_UNIQUE_SWITCH_OR_SOURCE
  for all pipe || ( count( a switch_or_source | (SWITCH union SOURCE) ||
                          ( pipe = switch_or_source.Switch_Outlet or
                            pipe = switch_or_source.Source_Outlet ) ) = 1 );
```

Gist comment - the values of attributes can be retrieved in the following manner: if *obj* is an object of type T, where type T has an attribute **Att**, then *obj*.**Att** denotes any object serving as *obj*'s **Att** attribute. end comment

```
relation IMMEDIATELY_BELOW(ib1 | LOCATION, ib2 | LOCATION)
  definition
    ib1 = ( case ib2 of
              a pipe =>ib2 : Pipe_Outlet;
              a switch =>ib2 : Switch_Outlet;
              the source =>ib2 : Source_Outlet
            end case );
```

Gist comment - the predicate of a defined relation denotes those tuples of objects participating in that relation. For any tuple of objects of the appropriate types, that tuple (in the above relation, a 2-tuple of LOCATIONS) is in the defined relation if and only if the defining predicate equals true for those objects. end comment

```
relation BELOW(b1 | LOCATION, b2 | LOCATION)
  definition
    IMMEDIATELY_BELOW(b1,b2) or
    ( exists b3 | LOCATION || ( BELOW(b1,b3) and BELOW(b3,b2) ) );

always required SOURCE_ON_ROUTE_TO_ALL_BINS
  for all bin || BELOW(bin, the source);
```

Packages--the objects moving through the network

```
type PACKAGE(At | LOCATION, Destination | BIN);

always prohibited MULTIPLE_PACKAGES_AT_SOURCE
  exists package.1, package.2 || ( package.1 : At = the source and package.2 : At = the source );
```

Our Portion

Spec comment - the portion over which we have control and are to implement end comment

```
agent PACKAGE_ROUTER() where

relation PACKAGES_EVER_AT_SOURCE(packages | sequence of PACKAGE)
 definition packages = ( a package ) ordered wrt package : At = ( the source ) ;
```

The source station

```
demon RELEASE_PACKAGE_INTO_NETWORK[package.new]
 trigger package.new : At = the source
 response
  begin
   if package.new : Destination ≠
    ( a package.previous ||
      package.previous immediately < package.new
                    wrt PACKAGES_EVER_AT_SOURCE(*) ) : Destination
   then WAIT[] ;
```

Spec comment - the demon must delay release of the new package if its destination differs from that of the previous package (the immediately preceding package to have been at the source). end comment

```
   update : At of package.new to (the source) : Source_Outlet
  end ;
```

Gist comment - a demon is a data-triggered process. Whenever a state change takes place in which the value of the demon's trigger predicate changes from false to true, the demon is triggered and performs its response. end comment

```
action WAIT[] ;
```

The switches

```
relation SWITCH_IS_EMPTY(switch)
 definition not exists package || ( package : At = switch ) ;

demon SET_SWITCH[switch]
 trigger Random()
 response
  begin
   require SWITCH_IS_EMPTY(switch);
   update : Switch_Setting of switch to switch : Switch_Outlet
  end;
```

Spec comment the nondeterminism of when and which way to set switches is constrained by the always prohibited that follows shortly. end comment

```
relation PACKAGES_TO_BE_ROUTED_BY_SWITCH(packages | sequence of PACKAGE, switch)
 definition
   packages =
     ( a package || ( BELOW(switch,package : At) and
                      BELOW(package : Destination,switch) and
                      not PACKAGE_IN_CORRECT_SWITCH_WRONGLY_SET(package) )
     ) ordered wrt PACKAGES_EVER_AT_SOURCE(*) ;
```

Spec comment - packages to be routed by a switch are those packages for whom the following conditions are true. (i) the switch lies below them, (ii) the switch lies on their routes to their destinations; and (iii) they are not in some switch set the wrong way. The sequence is ordered by the order in which the packages were at the source Note that this excludes packages that are already misrouted; there may be such packages on their way to this switch, but since they are already misrouted the switch will not have to route them in any particular direction. end comment

```
relation PACKAGE_IN_CORRECT_SWITCH_WRONGLY_SET(package)
 definition
  exists switch || ( package : At = switch and
                     BELOW(package : Destination,switch) and
                     not BELOW(package : Destination,switch : Switch_Setting) ) ;
```

Spec comment - a package is in a correct switch that is wrongly set if the switch lies on the route to that package's destination but the switch is currently set the wrong way. (This is how a package becomes misrouted.) end comment

```
always prohibited DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE
 exists package, switch ||
  ( PACKAGE_IN_CORRECT_SWITCH_WRONGLY_SET(package)
   and
   ( ( SWITCH_IS_EMPTY(switch) and
       package = first(PACKAGES_TO_BE_ROUTED_BY_SWITCH(*,switch) ) )asof true )
  ) ;
```

Spec comment - there must never be a state in which a package is in a wrongly set switch if there has been an opportunity to set the switch correctly for that package, i.e., at some time that package was the first of those due to be routed by the switch and the switch was empty. end comment

Indicating the arrival of a misrouted package in a bin

```
demon DETECT_MISROUTED_ARRIVAL[package.misrouted, bin.reached, bin.intended]
 trigger package.misrouted : At = bin.reached and
         package.misrouted : Destination = bin.intended
 response DISPLAY_MISROUTING[ bin.reached, bin.intended ] ;

action DISPLAY_MISROUTING[ bin.reached, bin.intended ];
end
```

The Environment

agent ENVIRONMENT() where

Arrival of packages at the source

```
demon CREATE_PACKAGE[]
  trigger Random()
  response
    create package.new || ( package.new : Destination = a bin  and
                            package.new : At = the source ) ;
```

Spec comment - for the purposes of defining the environment in which the package router is to operate, packages with some random bin as their destination appear at random intervals at the source (subject to the prohibition on there being multiple packages at the source). end comment

Movement of packages through the network

```
relation CONNECTED_TO(location.1, location.2)
  definition
    location.2 = ( case location.1 of
                     a pipe => location.1 : Pipe_Outlet;
                     a switch => location.1 : Switch_Setting
                   end case );
```

```
demon MOVE_PACKAGE[package, location.next]
  trigger Random()
  response
    if CONNECTED_TO(package : At, location.next)
    then update : At of package to location.next;
```

Spec comment - modelling the unpredictable movement of packages through the network is achieved by having this demon at random move a random package from one location to the next CONNECTED_TO location end comment

```
always prohibited MULTIPLE_PACKAGES_REACH_LOCATION_SIMULTANEOUSLY
  exists package.1, package.2, location
    || ( ( start package.1 : At = location ) and ( start package.2 : At = location ) ) ;
```

Gist comment - start <predicate> is true if the predicate has just changed in value from false to true. end comment

Spec comment - the mechanical construction of the router is such that, although packages may bunch up, the passage of each individual package may be detected. This we model by constraining the "granularity" of movement to be that of individual packages. end comment

always prohibited PACKAGES_LEAVE_OUT_OF_ORDER
  exists *package.1, package.2, common_location* | LOCATION
    || ((start *package.1* : At = *common_location* <
      start *package.2* : At = *common_location*) and
      package.1 : At = *common_location* and
      not *package.2* : At = *common_location*);

Spec comment - it is prohibited that package 1 enter before package.2 yet still be there when package.2 has left. end comment

Abstraction Specification

Gist comment - the behaviors denoted by the specification are an abstraction of the detailed behaviors of the preceding system. This section states just which of those details are to be included in the abstracted behaviors. end comment

abstraction
  types
    LOCATION, SOURCE, PIPE, SWITCH, BIN, PACKAGE ;

  attributes
    Source_Outlet, Pipe_Outlet, Switch_Outlet, Switch_Setting,
    At, Destination ;

  actions
    DISPLAY_MISROUTING, WAIT
end abstraction

Implementation Specification

Gist comment - this section states what the portion to be implemented may observe and what it may effect. end comment

```
implement PACKAGE_ROUTER
  observing
    types
      LOCATION, SOURCE, PIPE, SWITCH, BIN, PACKAGE;

    attributes
      Source_Outlet, Pipe_Outlet, Switch_Outlet, Switch_Setting :

    predicates
      start ( a package ) : At = the source,
      start ( a package ) : At = switch,
      finish ( a package ) : At = switch,
      start ( a package ) : At = bin ;
      exists package || ( package : Destination = bin and package : At = the source ) ;
```

Spec comment - the router is limited to observing the routing network, the arrival of packages at the source and their movement into and out of switches and into bins, and the destination of a package while it is located at the source. end comment

```
  effecting
    attributes
      switch : Switch_Setting,
      package : At asof package : At = the source ;
```

Spec comment - the router is limited to effecting the setting of switches and the release of packages at the source. end comment

```
end implement
```

REFERENCES

1. Balzer, R., "Transformational Implementation: An example," *IEEE Transactions on Software Engineering* SE-7, (1), 1981, 3-14.

2. Balzer, R., and N. Goldman, "Principles of good software specification and their implications for specification languages," in *Specification of Reliable Software*, pp. 58-67, IEEE Computer Society, 1979.

3. Balzer, R., N. Goldman, and D. S. Wile, "On the transformational implementation approach to programming," in *Proceedings of the Second International Conference on Software Engineering*, pp. 337-344, San Francisco, October 1976.

4. Balzer, R., N. Goldman, and D. S. Wile, "Informality in program specifications," *IEEE Transactions on Software Engineering* SE-4, (2), 1978, 94-103.

5. Bauer, F. L., "Programming as an evolutionary process," in *Proceedings of the Second International Conference on Software Engineering*, pp. 223-234, IEEE, San Francisco, 1976.

6. Bauer, F. L., H. Partsch, P. Pepper, and H. Wössner, *"Notes on the Project CIP: Outline of a Transformation System"*, institut für Informatik, Technische Universität München, Technical Report TUM-INFO-7729, 1977.

7. Bauer, F. L., M. Broy, R. Gnatz, H. Partsch, P. Pepper, and H. Wössner, "Towards a wide spectrum language to support program specification and program development," *SIGPLAN Notices* 13, (12), December 1978, 15-23.

8. Birtwistle, G. M., O. Dahl, B. Myhrhaug, and K. Nygaard, *SIMULA Begin*, Auerbach, 1973.

9. Bobrow, D., and B. Raphael, "New programming languages for artificial intelligence," *ACM Computing Surveys* 6, (3), September 1974, 153-174.

10. Burstall, R. M., and J. Darlington, "A transformation system for developing recursive programs," *Journal of the ACM* 24, (1), January 1977, 44-67.

11. Burstall, R. M., D. B. MacQueen, and D. T. Sannella, "HOPE: An experimental applicative language," in *Proceedings of the 1980 LISP Conference*, pp. 136-143, Stanford, 1980.

12. Darlington, J., *A Semantic Approach to Automatic Program Improvement*, Ph.D. thesis, University of Edinburgh, Department of Artificial Intelligence, 1972.

13. Darlington, J., "Applications of program transformation to program synthesis," in *Proceedings of the International Symposium on Proving and Improving Programs*, pp. 133-144, Arc-et-Senans, France, 1975.

14. Darlington, J., "A synthesis of several sorting algorithms," *Acta Informatica* 11, (1), December 1978, 1-30.

15. Darlington, J., "An experimental program transformation and synthesis system," *Artificial Intelligence* 16, (1), March 1981, 1-46.

16. Darlington, J., and R. M. Burstall, "A system which automatically improves programs," *Acta Informatica* 6, (1), March 1976, 41-60.

17. Dewar, R. B. K., A. Grand, S. Liu, and J. T. Schwartz, "Programming by refinement, as exemplified by the SETL representation sublanguage," *ACM Transactions on Programming Languages and Systems* 1, (1), 1979, 27-49.

18. Doyle, J., "A truth maintenance system," *Artificial Intelligence* 12, (3), 1979, 231-272.

19. Earley, J., "High level iterators and a method for automatically designing data structure representation," *Computer Languages* 1, (4), 1975, 321-342.

20. Feather, M. S., *"ZAP" Program Transformation System Primer and Users Manual*, Department of Artificial Intelligence, University of Edinburgh, Technical Report DAI 54, 1978.

21. Feather, M. S., and P. E. London, "Implementing specification freedoms," *Science of Computer Programming* 2, 1982, 91-131.

22. Feather, M. S., "A system for assisting program transformation," *ACM Transactions on Programming Languages and Systems* 4, (1), January 1982, 1-20.

23. Fickas, S. F., "Automatic goal-directed program transformation," in *Proceedings of the First Annual National Conference on Artificial Intelligence*, pp. 68-70, AAAI, Stanford, August 1980.

24. Fickas, S. F., *Automating the Transformational Development of Software*, USC/Information Sciences Institute, RR-83-108 and RR-83-109, 1983. (Published in two volumes.)

25. Goldman, N., and D. S. Wile, "A relational data base foundation for process specification," in P. Chen (ed.), *Entity-Relationship Approach to Systems Analysis and Design*, North-Holland, 1980.

26. Hommel, G., *Vergleich Verschiedener Spezifikationsverfahren am Beispiel Einer Paketverteilanlage*, Kernforschungszentrum Karlsruhe, Technical Report, August 1980.

27. Hopcroft, J. E., and J. D. Ullman, *Formal Languages and their Relation to Automata*, Addison-Wesley, 1969.

28. London, P. E., "A dependency-based modelling mechanism for problem solving," in *AFIPS Conference Proceedings, Vol. 47: National Computer Conference*, pp. 263-274, 1978.

29. Manna, Z., and R. Waldinger, "Synthesis: Dreams => programs," *IEEE Transactions on Software Engineering* SE-5, (4), 1979, 294-328.

30. McDermott, D., and G. J. Sussman, *The CONNIVER Reference Manual*, MIT, Memo 259a, 1974.

31. Neighbors, J. M., *Software Construction Using Components*, Ph.D. thesis, University of California, Irvine, 1980.

32. Paige, R., and S. Koenig, "Finite differencing of computable expressions," *ACM Transactions on Programming Languages and Systems* 4, (3), July 1982, 402-454.

33. Paige, R., and J. Schwartz, "Expression continuity and the formal differentiation of algorithms," in *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pp. 58-71, Los Angeles, 1977.

34. Schonberg, E., J. T. Schwartz, and M. Sharir, "An automatic technique for selection of data representations in SETL programs," *ACM Transactions on Programming Languages and Systems* 3, (2), April 1981, 126-143.

35. Schwartz, J. T., *On Programming, an Interim Report on the SETL Project*, New York University, Courant Institute of Mathematical Sciences, Technical Report , June 1975.

36. Sernadas, A., "Temporal aspects of logical procedure definition," *Information Systems* 5, (3), 1980, 167-187.

37. Smith, J., and D. Smith, "Database abstractions: Aggregation and generalization," *ACM Transactions on Database Systems* 2, (2), 1977, 105-133.

38. Wile, D. S., *Program Developments as Formal Objects*, USC/Information Sciences Institute, RR-82-99, 1982.

39. Wile, D. S., *POPART: Producer of Parsers and Related Tools, System Builders' Manual*, USC/Information Sciences Institute, 1983. (In press.)

40. Zave, P., "An operational approach to requirements specification for embedded systems," *IEEE Transactions on Software Engineering* SE-8, (3), May 1982, 250-269.

41. Zave, P., and R. T. Yeh, "Executable requirements for embedded systems," in *Proceedings of the Fifth International Conference on Software Engineering*, pp. 295-304, IEEE Computer Society Press, San Diego, 1981.